Introduction to Fortran 90, Part IV

CS 471/Math 471, Fall 1997

Brian T. Smith, UNM

Andrew C. Pineda, UNM

Pointers In Fortran

- Strongly typed
- Tightly specified to permit traditional optimizations of non-pointer code
  - requires the use of the TARGET attribute for any variable that can be pointed to
- No pointer arithmetic allowed
- Automatic de-referencing in non-pointer contexts

Automatic De-Referencing

- Use of the pointer is a reference to its target in most cases rather than its pointer
  - in an expression
  - as the variable on the left side of an assignment
  - in an I/O list
  - as an actual argument in an argument list, unless the corresponding dummy argument has the pointer attribute

Pointer Examples

```
real, pointer :: A_PTR, B_PTR
real, target :: A = 3.1, B = 4.2
integer, pointer :: CP
integer, target :: C = -32

A_PTR => A;  B_PTR => B; CP => C
A_PTR = B_PTR + C     ! Targets referenced --
                      ! A_PTR becomes -27.8.
CP => A   ! Invalid
CP = sqrt(A_PTR + B_PTR) + abs(CP)  ! Targets
```


Pointer Model

- A variable with the pointer attribute contains either:
  - an address of a target (associated with a target)
  - null value (disassociated, points to null)
  - undefined (unpredictable, not testable)
- Initially, a pointer is in the undefined state

Disassociated State

- A pointer becomes disassociated by being:
  - nullified as in the statement
    `nullify( PTR )`
  - deallocated as in the statement
    `deallocate( PTR )`
  - pointer assigned to a disassociated pointer
    ```
    nullify( A_PTR )
    PTR => A_PTR
    ```

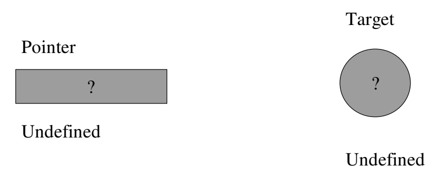
Associated Pointer

- A pointer becomes associated by being:
  - pointer assigned to a target or to a pointer that is associated with a target

    A_PTR => A ! Assume A is a target
  - allocated using

    allocate(A_PTR)

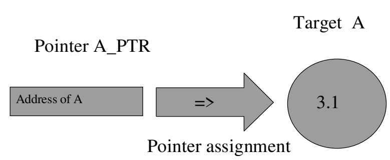
Initial States



Associated States

- After

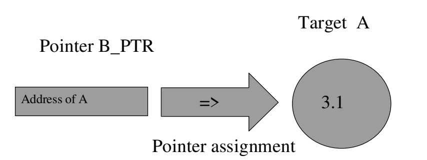
  A = 3.1; A_PTR => A



Associated States Cont'd

- After

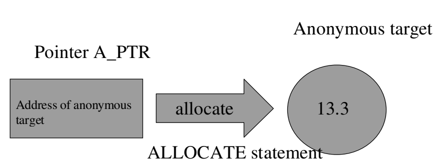
  A = 3.1; A_PTR => A; B_PTR => A_PTR



Associated States Cont'd

- After

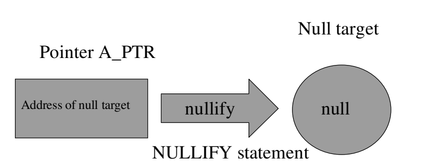
  allocate(A_PTR); A_PTR = 13.3



Disassociated State

- After

  nullify(A_PTR)



Examples of Pointers

```
real, pointer :: PTR
real, target :: A, B, C

PTR => A          ! Pointer assignment.
                  ! PTR now has A as its target.
PTR => B          ! Change the target of PTR to B.
allocate( PTR )   ! Create storage for a target and point
                  ! PTR to this storage (target is
                  ! anonymous, has no name).
```


Rules For Pointer Assignment

- Form
  <pointer> => <target>
  - <pointer> and <target> must have the same type, kind, and rank
  - <target> must be a variable or function with the TARGET or POINTER attribute
  - <pointer> must be a variable with the POINTER attribute

Rules For Reference To A Pointer

- In a pointer assignment, <pointer> is a reference to the address of the target
- In an argument list corresponding to a dummy argument with the POINTER attribute, it is the address of the target
- In all other executable constructs, it is a reference to the target

Major Uses of Pointers in SComp

- Avoiding the copy operation for large data structures
- Dynamically size data structures
  - arrays
  - lists of objects of arbitrary size
  - sparse matrices

Avoiding Large Copy Operations

- Iterating with a large grid of points with a large amount of data associated with the grid points (pressures, temps, velocities, etc)
  - may be megabytes of storage
    ```
    OLD_GRID = NEW_GRID
    NEW_GRID = ... OLD_GRID ...
    ```
  - first assignment moves lots of data

No Data Movement With Pointers

```
T => OLD_GRID
OLD_GRID => NEW_GRID
NEW_GRID => T
NEW_GRID = ...  OLD_GRID ...
```

- No data is moved when => is used
- The computation of the new grid values are stored in the location for NEW_GRID which was the location of the OLD_GRID on the previous iteration

Interfaces

- An interface is the set of characteristics of a procedure needed to ensure a correct reference to the procedure. The characteristics include:
  - the name of the procedure
  - the kind of procedure (function or subroutine)
  - the type, kind, rank, and other attributes of the returned result, if a function
  - the number and order of the arguments
  - the name, type, kind, rank, and other attributes of the arguments
- Interfaces are either explicit or implicit

Implicit Interfaces

- Are those that are assumed from the reference site from the name, arguments, and form of the reference
- Interfaces are implicit when the reference is to an:
  - external procedure with no interface block visible
  - statement functions

Explicit Interfaces

- Are those that are known completely at the site of the reference
- They include references to:
  - internal and module procedures
  - intrinsic procedures
  - external procedures with interfaces specified by a visible interface block

Explicit Interfaces Required

- When the referenced procedure
  - has optional arguments
  - is an array-valued function
  - is a pointer-valued function
  - is a character-value function with dynamic length
  - has assumed-shape array arguments
  - has arguments with TARGET or POINTER attr

Explicit Interfaces Req'd Cont'd

- When the procedure:
  - is called with keyword actual arguments
  - is called with a generic name
  - is called via a user-defined operator
  - is called via a user-defined assignment
- Interface blocks specify an interface for external or dummy procedures

Interface Blocks

- Form (non-generic):

```
interface
   <subprogram_header>
      <specification_part_for_dummy_arguments>
   <end_statement_for_subprogram>
       … Zero_or_more_of_the_above_3_items_repeated …
end interface
```

  - Example

```
interface
   real function my_sqrt( a); real, intent(in) :: a; end function
end interface
```

Interface Blocks Continued

- Form (generic name, operator, assignment)

```
interface  name |  operator( <op> ) |  assignment(=)
    <interface_block_as_on_previous_slide>
    or
    module procedure  <list_of_one_or_more_names>
end interface
```

  – Example

```
interface  operator( .sqrt. )
  module procedure  my_sqrt
end interface
```


Modules

– A program unit that collects together objects to be shared by specified program units
  - objects include:
    – variables, named constants, derived types, module procedures, interfaces, generic identifiers, namelist groups

– Sharing is determined by use of
  - PUBLIC and PRIVATE attributes of objects in the module
  - USE statement in specified program units

Module Specification Part

– Declares all objects except module procedures to be shared. Specifies:
  - the usual attributes (type, rank, kind, parameter…)
  - accessibility attributes (PRIVATE, PUBLIC)

– Module procedure part specifies the interfaces for module procedures in:
  - module procedure header
  - module procedure specification part

Accessibility Attributes

- PRIVATE statement

  PRIVATE
  - sets the default accessibility attribute of all objects to PRIVATE

  PRIVATE <list_of_objects>
  - sets the accessibility of the listed objects to PRIVATE

- PUBLIC statement is analogous

Accessibility Attributes Cont’d

- Accessibility can be specified as an attribute of the object in its type declaration

```
real, dimension(3,4), private ::  A, B
integer, parameter, public :: WP = selected_int_kind(6)

type, private :: sparse_matrix
  real, pointer, dimension(:) ::  nonzeros
  integer, pointer, dimension(:) :: subscr1, subscr2
end type
```


Objects With Accessibility

- Names of:
  – variables, named constants, derived types, procedures, namelist groups, generic names
- Operators:
  – operator(<operator_name>)

    private operator(.inverse.)   ! Computes $A^{-1}b$

  – assignment(=)

    private assignment(=)   ! Say assigns rationals

Access To A Module

- Access to objects in a module is specified by a USE statement in a particular program unit where access is desired
- Forms (alternatives):
  - USE <module_name>
  - USE <module_name>, only : <list_of_objs_and_renames>
  - USE <module_name>, <list_of_renames>

Renames and Examples Of USE Statements

- Renames have the form:
  - <user_name> => <module_name>
- Examples:
  - USE sqrt_mod ! Accesses all public entities
  - USE sqrt_mod, only : sqrt => my_sqrt
    - ! Accesses only my_sqrt and renames it as sqrt

MY_SQRT In A Module

- Uses a module to specify working precision
  - allows one to change the value of WP once, recompile the program, and the precision of all data objects and prodecures are changed
- Uses a module to contain the function my_sqrt; defines it as a module procedure
  - taken intact from the internal procedure in the internal procedure example (see sqrtmod.f90)

Working Precision Module

```
module  WORKING_PRECISION
  ! This module defines a named constant
  ! that specifies the kind number for
  ! my working precision.

  implicit  none
  intrinsic  selected_real_kind
  integer, parameter :: WP = &
                selected_real_kind( 6 )
end module  WORKING_PRECISION
```


Module For My Square Roots

```
module MY_SQUARE_PACKAGE
  ! This module defines functions
  ! that compute approximations to
  ! square roots.
  ! Import the kind number WP for the
  ! working precision.
  use  WORKING_PRECISION, only : WP
    implicit  none

  contains
```


Module Continued

```
real(WP) function MY_SQRT( A )
   implicit  none
   real(WP), intent(in) :: A
   real(WP)  LOC_A, PREV_X, X
   real(WP), parameter :: FUDGE = &
                               2.0_WP
   intrinsic  abs, epsilon
   LOC_A = abs(A)
   X = LOC_A
   PREV_X = 0.5_WP*X
```

Module Continued

```
if( A /= 0.0_WP )  then
  do while( abs( X - PREV_X ) > &
        FUDGE*abs(X)*epsilon(X) )
    PREV_X = X
    X = 0.5_WP*(X + LOC_A/X)
  end do
else
  X = 0.0_WP
endif
MY_SQRT = X
end function  MY_SQRT
end module MY_SQUARE_PACKAGE
```


Module With Generic Overload

- See the file sqrtarr.f90
- Changed specification part
  - accessibility

```
private
intrinsic  abs, epsilon
real(WP), parameter :: FUDGE = &
                         2.0_WP
real(WP) :: EPS = FUDGE* &
                   epsilon( FUDGE )
```


Generic Overloads Cont'd

- Interface specification

```
! Provide an interface so that the
! function MY_SQRT becomes generic,
! allowing its argument and result to
! be either scalar or a 1-D array.

public  MY_SQRT
interface  MY_SQRT
module procedure  MY_SQRT_SCALAR,
  MY_SQRT_1D
end interface
```


1-D Array Module Procedure

```
function MY_SQRT_1D( A )
  implicit  none
  real(WP), intent(in), dimension(:)&
                                   :: A
  real(WP), dimension( size(A) ) :: &
                          MY_SQRT_1D
  real(WP), dimension( size(A) ) :: &
                   LOC_A, PREV_X, X
  LOC_A = abs(A)
  X = LOC_X
  PREV_X = 0.5_WP*X
```


1-D Array Module Proc. Cont'd

```
do while( any(abs( X - PREV_X ) > &
                EPS*abs(X)) )
  where( X /= 0.0_WP )
    PREV_X = X
    X = 0.5_WP*(X + LOC_A/X)
  endwhere
end do

MY_SQRT_1D = X

end function  MY_SQRT_1D
```